**Patent Office
Canberra**

I, JULIE BILLINGSLEY, TEAM LEADER EXAMINATION SUPPORT AND SALES hereby certify that annexed is a true copy of the Provisional specification in connection with Application No. 2003901968 for a patent by WOLFGANG FLATOW and WELCON IT SOLUTIONS as filed on 23 April 2003.

I further certify that the name of the applicant has been amended to WOLFGANG FLATOW pursuant to the provisions of Section 104 of the Patents Act 1990.

WITNESS my hand this
Fifth day of May 2004

JULIE BILLINGSLEY
TEAM LEADER EXAMINATION
SUPPORT AND SALES

Patent for

A Universal Database Schema

Inventor: Wolfgang Flatow
12 Alfred Street
Tannum Sands
QLD 4680

Table of Contents

Patent for 'A Universal Database Schema'

A 'database schema' is the name of the overall database design, including tables and fields, relationships and indexes.

A 'universal schema' is a term coined to mean that the schema is highly flexible in how it can be used, and that schema changes are not required to store new classes of data.

A 'standard schema' uses a table for each class of data to be stored (ie. A Clients table with First Name, Last Name, Phone, Address etc fields). For each new class of data, a new table is required, with consequent schema changes.

This patent describes a particular configuration of Universal Schema (or database design).

Database designing has always been driven by Application Software requirements. Database designing has been, for the most part, driven by business functions, structure and processes and not by a need for flexibility and consistency in storage architecture.

A database designed for one application would have very different design (schema) than another database designed for a different application.

Over the last three-four decades, millions of databases and application systems have been developed and are being used by individuals, businesses, organizations and governments around the world.

The schema presented here provides a model that could be used for every one of these applications, providing a vast array of advantages.

It is able to provide a truly Universal Data Storage and Definition Structure in a Building Block format. It has implications for a common data query language for all data classes not otherwise possible with standard schemas.

It is a complete Data Class Abstraction incorporating all of the latest paradigms, including hierarchical representation, storage of data change history and many-to-many correlation of all data classes and fields.

Merits of this Universal Schema over a Standard Schema

Hierarchical storage model for all data

The Windows™ Explorer / file system and Windows™ management console are prime examples of how useful it is to store data in a way that facilitates hierarchical representation. When using standard schemas, it is not easy to combine all your tables in a single hierarchy.

This Universal Schema enables hierarchical data storage at its core.

A consistent storage model for all data

When data is stored on a 'Table per Data Class' basis, as is the case with standard schemas, there is no standard way to retrieve data – that is – for each table a distinct SQL statement and code is required that fits the table in question in order to interact with that data.

This universal schema provides a consistent method of interacting with all data, including hierarchical, field and relationship aspects.

Maximization of code re-use

The consistent storage model maximizes potential for code re-use. This can be achieved at all layers of an n-tier architecture, especially the database services layer.

The Forms section of this Universal schema further supports code re-use, to the extent that a single code body can provide viewing, editing and processing facilities for all current and future data classes.

New Data Classes can be introduced, changed or deleted without schema change

It is well known that database schema changes account for a significant percentage of development overheads. Estimates in my experience range from 40% to 75%.

In a standard schema, every single new table and/or field introduction or modification in a database results in a schema change. Every new data class you wish to introduce in a standard schema will require a schema change.

In order to appreciate this point, you can think of the schema as the foundations of a house and the application(s) built on the schema as the house. Every schema change is a foundation change requiring the house above to be adjusted/enhanced. The more of this goes on the more of a mess will become of the walls, wiring and plumbing etc.

In this Universal Schema, schema changes are not required to introduce, change or delete data classes.

Any field can store multiple values

A standard schema might have a client table with a phone number column or field. If you want to store 2 phone numbers you can either have an extra field (leaving you with 2) or a whole new table providing a multiple phone number facility providing any number of phone numbers.

The requirement for any N Multiple fields is very common, so in standard schemas, you need to create dedicated tables for each multiple field and the SQL and code to handle it.

In this Universal schema any field can be switched between a single or multiple field by setting a database Boolean (ie. again, no schema change)

Any field can store a history of changes

A common requirement for databases is to maintain a history of values for fields. Usually the history includes the date of the change and the person who changed it.

To achieve this functionality with a standard schema is very tediuous, and would involve 'table meta data tables' and a change log table. Further, it would involve elaborate sql and code to maintain.

This Universal Schema provides a consistent model for the storage and maintenance of history data without the need of meta data and change log tables.

A direct relationship may be created between any data classes without schema change.

Relationships are the glue that hold relational databases together. Whenever new tables or relationship fields are created in a standard schema, additional relationships must also be created to bind the tables together and to provide vital data integrity services from the database server.

Adding, changing or deleting a relationship is also a-schema change!

This Universal Schema allows relationships to be created, edited or deleted without a schema change.

Extensive Meta Data is available for all classes of data.

It is a very common requirement to associate 'Meta Data' (descriptive or functional data) with any given field in a table, or any given table within a database.

Examples of meta data for a field may be:

Name (the field name within databases is a restricted system name)
Description
Format
Read only
Required
Edit Control
Etc

In a standard database schema there is no inherent method of storing such meta data (other than may be provided by specific database servers).

In this Universal Schema extensive meta data is built in for every data class and field.

While a change of schema is required to add meta data to fields in the current version, this is at a different scope to the data and can therefore be managed easily (especially if meta data is added – not deleted). A method of introducing meta data without schema change is under development.

Conventions used in this patent

Table & Column (field) Names

All Table & Column names are expressed without spaces and are capitalised at the beginning of each component word. Ie. EntityType or ValueDate.
The names are designed to describe the intended function of the table or column.

Relationship columns

Where a column points to the unique identifier column of another table, that column name will consist of or end in the related tables name followed by 'ID'. Ie EntityTypeID or ValueEntityTypeID.

This convention defines the table relationships in this patent.

All Tables use an Identity Column called "ID".
While the sample schema shows a 4 byte LONG INTEGER numeric ID column, it is also acceptable to use an 8 BYTE Curency or 16 BYTE GUID (Globally Unique Identifier).

Required Columns

Each table described will define a set of required columns. These columns form the core configuration to enable and optimise the stated features and claims of this patent

Every table must have and identity column called 'ID' which is not shown for each table below.

Desired Columns

Each table described may define a set of desired columns. These columns optimise and/or extend the stated features of this patent

Optional Columns

Each table described may define a set of optional columns.

Entity Resources

We use this term to describe the Field and Value records that have been defined for an entity, and that must exist after an entity record is created.

Describing this Universal Schema

The sample schema diagram.

The associated schema diagram is titled "**Universal Schema V2.8**" and is marked as Copyright © 2003 Wolfgang Flatow & Welcon IT Solutions (21st of March 2003).

This schema definition is supported by a printout of a sample schema (as used by the prototype). The database server is Microsoft SQL Server 2000 ™.

The Table & Column names closely match those used in this patent, but are not identical. This patent covers the Tables & Columns as defined below, and their relationship as shown on the diagram.

Tables shown on the diagram and not defined in this patent are:

1. WebSession
2. Log
3. LogType

These tables provide optional Web and logging services for this schema.

The Data Storage System

This Universal Schema can manage all data with a simple Entity / Field / Value structure, where the field functions as the glue between the Entity, a FieldType that defines the field and its Value.

Storage efficiency is achieved by providing a table for each Data Type that requires storage. These are the Value Tables, one for each data type, with a pointer to their corresponding field. There is no additional burden on the schema or performance when adding Value tables, as the value tables point to the field and will only be requested if they exist.

The schema 'knows' what *entity resources* are required using the EntityType / FieldType structure, where there is a FieldType record defined for every field that an entity requires.

The schema 'knows' what Value table to use for what field by the FieldType / ValueType / DataType structure, where the DataType table points to corresponding Value Tables by name.

When referring to the schema diagram the names of the tables are the same as those below.

The Entity Table

As the name implies – any entity or anything at all that you may wish to store is represented by the entity. It provides the common interface to all items stored within this schema.

Required Columns:
- **EntityTypeID**
- **ParentEntityID** (Self referencing pointer)

Desired Columns:
- **Name**
- **Ordering**
- **EntitySatusID**
- **CreationDateTime**

Optional Columns:
- **Locked**

- **LastModified**
- **Expanded**

The Entity table is a self referencing hierarchical table that provides the 'skeleton' of the data structure.

The Entities corresponding definition (or meta data) table is the 'Entity Type' table.

An 'Entity Status' Table provides lifetime control such as created, active, deleted, archived etc.

A relationship table provides direct relationships not defined in the Entities hierarchical structure, by providing a FromEntity and a ToEntity pointer

The Field Table

In the sense that the Entity represents a table, the field represents the Column of a table in a standard schema.

The field is strictly glue between the entity, field type, and corresponding value of the field. Other than the IsDeleted, LastModified and ordering attributes, the field stores no data on its own.

Required Columns:
- **EntityID**
- **FieldTypeID**

Desired Columns:
- **IsDeleted** (enable field history)
- **Ordering** (ordering of 'multiple filelds')

Optional Columns:
- **LastModified**

Its name (and all other field meta data) comes from its corresponding FieldType record.

The field can have 1 or more Value Table records pointing at it.

The Value Tables

Each Value Table has a corresponding entry in the DataType table. They may also be called the Data Tables.

Required Columns:
- **FieldID**
- **A Value Column** (see below)

Desired Columns:
- **IsCurrentValue**
- **DateTime**
- **UserEntityID**

The desired columns enable a history of values to be kept.

The Value columns may be of any data type available within a given database Server:

- Text Value (ValueShortText, ValueLongText)
- Boolean Value (ValueBoolean)
- Currency Value (ValueCurrency)
- Date/Time Value (ValueDate)

- Binary Large Object Value (ValueBLOB)
- Entity Pointer Value (ValueEntity, ValueEntityType, ValueForm etc)
- Other Value

Note: These could also be called DataBoolean, DataDate etc.

The Data Definition Structure

The EntityClass Table

This is an optional table, and provide grouping and hierarchical services for the EntityType Table.

Required Columns:
- **Name**
- **ParentEntityClassID** (self referencing)

Optional Columns:
- **Description**

The EntityType Table

In the same way that the Entity table is a self referencing hierarchical table, the Entity Type table is also a self referencing hierarchical table. It defines the mandatory or default hierarchical structure of its corresponding entities.

The Entity Type provides the 'skeleton' of the data structure definition..

Required Columns:
- **Name**
- **ParentEntityTypeID** (self referencing)

Desired Columns:
- **EntityClassID** (pointer to the EntityClass table)
- **Ordering** (provide entity ordering between child entities)
- **IsFolder** (True if the entity is a 'container' for other entities)
- **IsCreatable** (can users create this type of entity?)
- **IsFixed** (can this only exist under my parent type?)
- **IsDefinitionEntity** (see below)
- **Help**

Optional Columns:
- **NodePrefix** (used in populating treeviews)
- **HasProcessForms** (used to determine processing forms)
- **Enum** (generate enumerated values)

IsDefinitionEntity: This schema stores items normally stored in 'lookup tables' in the Entity Table. When defining and EntityType for this kind of function it should be labelled as a definition entity. This helps to clearly distinguish entities of this type from entities used for main data storage.

The FieldType Table

The FieldType table is the main entity resource definition, providing a record for every field / Value pair required by all Entities that have this FieldTypes EntityTypeID.

Required Columns:
- **Name**
- **EntityTypeID** (The EntityType this FieldType belongs to)
- **ValueTypeID** (The ValueType of this FieldType)
- **FormatID** (The display Format to be used)

- **ValueEntityTypeID** (Provides a dropdown filter of Entities by EntityType. Valid for ValueEntity fields only.)
- **ValueParentEntityID** (Provides a dropdown filter for children of an Entity. Valid for ValueEntity fields only)
- **IsLocal** (Will not be created at entity creation – see below)
- **IsMultiple** (More than 1 field of this FieldType may be created – see below.)
- **Inherits** (Inherit values from the creator parent entities fields, if it contains one or more fields of this fieldtype)
- **KeepHistory** (use the history capability of this schema when dealing with fields of this FieldType)

Desired Columns:
- **Ordering**
- **InputRequired**
- **DefaultValue**
- **MinimumValue**
- **MaximumValue**
- **IsEntityDefault**
- **Help**

Optional Columns:
- **IsFixedMultiple** (Defines an Multiple Field with a fixed number of Multiples that are created on Entity creation. Works with MultiplesToCreate below)
- **MultiplesToCreate** (Sets the number of multiples to create for either IsMultiple or IsFixedMultiple FieldTypes.)

IsLocal: This defines a field for an entity that may or may not be required. Entity Resources marked as IsLocal do not get created when the entity is created. Some other factor determined by users or code may create or delete this field.

IsMultiple: This defines a FieldType that can have 0 to n Fields for any given Entity Resources marked as IsMultiple do not get created when the entity is created. Some other factor determined by users or code may create or delete any number of fields of this FieldType.

The ValueType Table

The function of the ValueType table is to "Type" the FieldType.

For example, you may have 2 Value Types – Date and Date/Time. They both store their data in the ValueDate table (by pointing to the ValueDate table entry in the DataType table – see below). While the storage location is the same for both Value Types, they are handled differently by code and edit dialogs, using the time component only for date/time value types.

Required Columns:
- **Name**
- **DataTypeID**

Desired Columns:
- **FormatID**

The DataType Table

The function of the DataType table is to link the FieldType/ValueType structure to the corresponding Value Table. There is one record in this table for every available Value Table. By including a TableName column (that stores the exact name of the corresponding Value Table) and a PrimaryDataField column (that stores the exact name of the corresponding data field) this can be used to switch in SQL and code to access different Value Tables and their data field.

Required Columns:

- Name
- TableName
- PrimaryDataFieldName

Desired Columns:
- DefaultFormatID
- DefaultControlID

Optional Columns:
- AccessName
- SQLServerName
- VBName

The optional columns provide some help for developers to identify the desired data types in different environments.

The Format Table

The format table stores a set of display formats for use by DataTypes, ValueTypes and FieldTypes.

Required Columns:
- Name
- Format
- DataTypeID

Composite Field Type

This is an optional table.

The purpose of this table is to allow more than 1 Value table to store a value for a given field.

This allows 1 value for each Value Table to be stored for each field, but not more than 1 of a given data type.

Required Columns:
- Name
- FieldTypeID
- ValueTypeID

Desired Columns:
- FormatID

The columns here may be built up to include most of the FieldType columns.

The EntityStatus Table

The EntityStatus table controls the lifecycle of an Entity.

This would typically include:

1. Created
2. Active
3. Withdrawn
4. Marked for Archiving
5. Marked for Deletion
6. Deleted

Required Columns:
- Name

Desired Columns:
- Description

The Relationship Table

This table provides relationships between entities that are not otherwise catered for by the Entity Hierarchy.

Required Columns:
* **RelationshipTypeID** (see below)
* **FromEntityID**
* **ToEntityID**

Desired Columns:
* **EntityFieldEntityID**

This provides a very versatile set of functionality in this schema. For example, if you need to perform processing on certain branches of the Entity Hierarchy, Relationships may be created to all parent nodes, which allows processing a branch without the need for iterating the Hierarchy, improving performance. It may also be used for any Grouping or Ownership relationships.

The RelationshipType Table

This table provides a 'Type' for the Relationship Table (see above) as well as providing EntityType, EntityClass and FieldType Relationship services.
It can define the type of entities connected using the Relationship Table and/or define relationships between EntityTypes, EntityClasses, FieldTypes and Entities.

Required Columns:
* **Name**
* **RelationshipGroupID**
* **FromEntityTypeID**
* **ToEntityTypeID**
* **FromEntityClassID**
* **ToEntityClassID**
* **FieldTypeID**
* **EntityID**

Desired Columns:
* **EntityFieldEntityID**

Optional Columns
* **Description**
* **Enum** (generate enumerators)

The RelationshipGroup Table

As the RelationshipType table can also be used to define relationships between definition tables, the RelationshipGroup table serves to group the RelationshipType Table entries.

Required Columns:
* **Name**

Optional Columns
* **Description**

The CompositeEntity Table

This table provides a very powerfull function within this schema – that of sharing or inheriting FieldTypes between EntityTypes.

For example, if numerous EntityTypes where participating in process control, then process control pointer fields can be created for a common 'process control EntityType' that may then be shared by any other EntityType by an entry in this CompositeEntity Table.
In this way an EntityTypes FieldTypes may be built up of several shared 'composite EntityTypes' and may or may not have any native FieldTypes itself.

Required Columns:
- **Name**
- **CompositeEntityTypeID**
- **FromSourceEntityTypeID**
- **ToTargetEntityTypeID**

Composite Entity Type

This table serves to group CompositeEntity records, and to provide a pointer to dedicated composite EntityTypes.
Required Columns:
- **Name**
- **SourceEntityTypeID**

Optional Columns
- **Description**

Forms

The forms system is designed to provide numerous display and processing services for this schema. It is, again, a highly flexible and generic structure that may be used for:

- Generate any number of views or forms of an Entities data
- Independent naming, formatting and controls display for each form
- Defining fields to display in browse lists
- Defining fields to display in reports

The Form Table

This table contains one record for each defined Form.
A form is usually associated with an EntityType that supplies the FieldTypes to display, but this is not a requirement. A single Form bay combine FieldTypes from any EntityTypes.

Required Columns:
- **Name**
- **FormTypeID**
- **DataEntityTypeID**
- **FormTabID**
- **IsDefaultForm** (default form for the DataEntityType)

Desired Columns:
- **TargetFormID** (Define form sequences)
- **SubmitButtonText** (If other than 'Submit')
- **ValidateForm**

Optional Columns
- **Description**

The FormType Table

This serves to group Forms and to define Form Function.
For example, you may have FormTypes of Edit, View, Browse, Report etc.

Required Columns:
- Name
- FormTabGroupID

The FormGroup Table

The FormGroup table provides grouping of FormField records for each Form. For example, in a Stock Item Form you might have a 'Details', a 'Suppliers' and a "Locations" group. FormGroups also serve to group one or more 'multiple FieldType' values.

Required Columns:
- Name
- FormID
- Ordering

Optional Columns
- Enum

The FormField Table

This defines the FieldTypes to display, their names, format and editing control.
It can easily be extended to contain default, minimum, maximum and validation information for each FormField.

Required Columns:
- Name
- FormID
- FormGroupID
- FieldTypeID
- ControlTypeID
- Ordering

Desired Columns:
- InputRequired
- ViewOnly

Optional Columns
- Enum

The ControlType Table

This defines the different editing or display controls that may be used to edit or output data. For example, it would contain an entry for text editing control, a checkbox, a radio button etc.

Required Columns:
- Name
- Width
- Height
- MaxChar

Desired Columns:
- NetscapeWidth
- XSL

The FormTab Table

These Tabs are the ones commonly shown along the top of a form to provide a link to other forms in a group.
This is used to define a set of tabs, grouped by FormTabGroupID fields.
A Form can then be associated with a FormTab

Required Columns:
- Name
- Image
- FormTabGroupID

The FormTabGroup Table

This serves to group FormTabs.

Required Columns:
- Name

Claims

Claim 1. The Universal Data Storage System

I claim the invention of the universal data storage system defined as:
Hierarchical **Entity** Table / **Field** Table / **Data** Tables
Where the Entity has a self referencing parent pointer that allows a hierarchy to be defined, where each Entity has zero or more Fields and each Field has at least one Data Table entry in one of the Data Tables.
Where there is one or more Data table for desired (or all available) Data Types and any desired pointers to other database objects (ie Entity, EntityType, Form etc).
Allowing more than one Field of a given FieldType for an Entity.
Allowing historical data to be stored in each Data Table for a given Field.

Claim 2. The Universal Data Definition System

I claim the invention of the data definition system to support the Universal Data Storage System in Claim 1, defined as:
Hierarchical **EntityClass** Table (optional) / Hierarchical **EntityType** Table / **FieldType** Table / **ValueType** Table / **DataType** Table
Where the EntityType has a self referencing parent pointer that allows a hierarchy to be defined, where each EntityType has zero or more FieldTypes and each FieldType has one ValueTypes, each of which has a DataType.
Providing a system of automated construction of Entity, Field and Data Records.
Providing extensive Meta (or descriptive) Data for each Entity and Field.
Providing a system of data class definition without the need for schema changes.

Claim 3. Combination of Claim 1 & Claim 2

I claim the invention of a Universal Data Storage & Definition Schema that combines Claim 1 & Claim 2.
The resulting schema having the capacity to define and store any conceivable data class (normally requiring a dedicated table) without schema changes.
Where Database 'Views' can be automatically generated for any of the EntityTypes defined in the schema that provide a 'standard' or 'flat' table from the universal schema.

Claim 4. The Universal Forms System

I claim the invention of the Forms Definition System defined as:
FormType Table / **Form** Table / **FormGroup** Table / **FormField** Table with a correlating **FormTabGroup** Table / **FormTab** Table structure.
Where each Form is of a given FormType and has n FormGroups, each with n FormFields.
Where each Form can be linked to a EntityType.
Where each FormField is linked to a FieldType.
Providing a system of defining any number of form views of an EntityTypes FieldTypes (with data from a corresponding Entities Fields).
Providing a system of defining any sets of FieldTypes for browsing, reporting etc purposes.
Providing a layer of abstraction that allows data labelling, display and processing to be defined for each Form.

Claim 5. Combination of Claim 3 & Claim 4

I claim the invention of a Universal Data Storage, Definition & Display Schema that combines Claim 3 & Claim 4.
The resulting schema provides a comprehensive range of services allowing complete applications to be built without need for schema changes.
The range of potential applications for this schema is claimed to be:
EVERY APPLICATION REQUIRING A DATABASE
That is, it is always an option to utilize this schema instead of designing a standard schema.

This schema also provides many opportunities for a powerful database services layer, including Entity creation and destruction, data access, hierarchical services etc.

Composite Entity Types provide
field definitions that can be re-used by
other Entity Types

Define Entities

EntityType
- ID
- Name
- NodePrefix
- Help
- EntityClassID
- StartingProcessEntit
- ParentEntityTypeID
- Ordering
- IsRequired
- IsFolder
- IsCreatable
- IsFixed
- IsDefinitionEntity
- HasProcessForms

CompositeEntityType
- ID
- Name
- SourceEntityTypeID
- CopyFieldDefinitions
- Description

CompositeEntity
- ID
- FromSourceEntityTypeID
- ToCompositeEntityTypeID
- CompositeEntityTypeID

**Define Storage Fields
for EntityTypes**

FieldType
- ID
- Name
- NodePrefix
- Help
- Icon
- Ordering
- FormatID
- EntityTypeID
- ValueEntityTypeID
- ValueParentEntityID
- ValueTypeID
- InputRequired
- Width
- MaxChar
- DefaultValue

RelationshipType
- ID
- Name
- RelationshipGroupID
- FromEntityTypeID
- ToEntityTypeID
- FromEntityClassID
- ToEntityClassID
- FieldTypeID
- EntityID
- Description
- OnCreateDestroy
- Enum

EntityClass
- ID
- Name
- RootEntityClassID
- Description

Relationship
- ID
- RelationshipTypeID
- FromEntityID
- ToEntityID
- EntityFieldEntityID

RelationshipGroup
- ID
- Name
- Description

CompositeFieldType
- ID
- Name
- FieldTypeID
- ValueTypeID

ValueType
- ID
- Name
- FormatID
- DataTypeID
- Minimum
- Maximum
- Description

Value Type
provides a layer of
abstraction for
FieldType

Format
- ID
- Name
- Format
- DataTypeID

DataType
- ID
- Name
- TableName
- DefaultFormatID
- DefaultControlTypeID
- AccessName
- SQLServerName
- VBName

Format provides
display formats for
all data types.

DataType controls
what Value Table
should be used

ïx

）
peID
ityTypeID
rentEntityID
peID
quired

ır
Value
nValue
nValue
;

rDefault
ıle
sToCreate
IMultiple
te
able
berRoleParty
ionsibleRoleParty
lstory
ByDate

Define Form Views for EntityTypes

WebSession
- ID
- UserGUID
- UserIP
- UserEntityID
- WebPageEntityID
- CurrentFormID
- CurrentEntityID
- IsAdministrator
- LastDateTime
- SessionValues
- JumpEntityID
- JumpLayer
- HighestMembershipPointID

Form
- ID
- IDName
- Name
- FormTypeID
- FormTabID
- DataEntityTypeID
- TargetFormID
- SubmitButtonText
- IsDefaultForm
- DoNotDestroy
- ValidateForm
- ProcessEntityID

ValueForm
- ID
- FormID
- FieldID
- DateTime
- UserEntityID
- IsCurrentValue

FormType
- ID
- Name
- FormTabGroupID

FormGroup
- ID
- Name
- FormID
- Ordering
- Enum
- PermissionEntityID

FormField
- ID
- Name
- InputRequired
- FormID
- FieldTypeID
- FormGroupID
- ControlTypeID
- Ordering
- ViewOnly
- Enum

FormTabGroup
- ID
- Name

FormTab
- ID
- Name
- Image
- FormTabGroupID
- EntityID

ControlType
- ID
- Name
- Width
- Height
- MaxChar
- XSL

Hierarchical Entity
Storage

Entity
- ID
- Name
- IsLocked
- Ordering
- ParentEntityID
- EntityTypeID
- EntityStatusID
- CreationDateTime
- CreationComputer
- CreationUserEntityID
- LastModified
- Expanded

EntityStatus
- ID
- Name
- Description

Entity Status is the life cycle of
the entity (created, active &

Universal Data Field
Storage System

ValueEntityType
- ID
- FieldID
- EntityTypeID
- IsCurrentValue

ValueDate
- ID
- DateVal
- FieldID
- UserEntityID
- DateTime
- IsCurrentValue

ValueBoolean
- ID
- BooleanVal
- FieldID
- UserEntityID
- DateTime
- IsCurrentValue

ValueNumber
- ID
- NumberVal
- FieldID
- UserEntityID
- DateTime
- IsCurrentValue

ValueBlob
- ID
- Blob
- FieldID
- UserEntityID
- DateTime
- IsCurrentValue

Field
- ID
- Ordering
- LastModified
- EntityID
- FieldTypeID
- IsDeleted

ValueCurrency
- ID
- CurrencyVal
- FieldID

ValueLongText
- ID
- LongTextValue
- FieldID

V.
to
cl

Universal Data Field Storage System

Field
- ID
- Ordering
- LastModified
- EntityID
- FieldTypeID
- IsDeleted

ValueBoolean
- ID
- BooleanVal
- FieldID
- UserEntityID
- DateTime
- IsCurrentValue

ValueNumber
- ID
- NumberVal
- FieldID
- UserEntityID
- DateTime
- IsCurrentValue

ValueBlob
- ID
- Blob
- FieldID
- UserEntityID
- DateTime
- IsCurrentValue

ValueLongText
- ID
- LongTextValue
- FieldID
- UserEntityID
- DateTime
- IsCurrentValue

ValueCurrency
- ID
- CurrencyVal
- FieldID
- UserEntityID
- DateTime
- IsCurrentValue

(table)
- gTypeID
- gEntityID
- eldID
- serEntityID
- ateTime
- omment

Type
- ID
- Name

ValueEntity
- ID
- EntityID
- FieldID
- UserEntityID
- DateTime
- IsCurrentValue

**Universal Data Field
Storage System**

Field
- ID
- Ordering
- LastModified
- EntityID
- FieldTypeID
- IsDeleted

ValueBoolean
- ID
- BooleanVal
- FieldID
- UserEntityID
- DateTime
- IsCurrentValue

ValueNumber
- ID
- NumberVal
- FieldID
- UserEntityID
- DateTime
- IsCurrentValu

ValueBlob
- ID
- Blob
- FieldID
- UserEntityID
- DateTime
- IsCurrentVah

ValueLongTe
- ID
- LongTextVal
- FieldID
- UserEntityID
- DateTime
- IsCurrentVal

Log
- ID
- LogTypeID
- LogEntityID
- FieldID
- UserEntityID
- DateTime
- Comment

ValueCurrency
- ID
- CurrencyVal
- FieldID
- UserEntityID
- DateTime
- IsCurrentValue

LogType
- ID
- Name

ValueEntity
- ID
- EntityID
- FieldID
- UserEntityID
- DateTime
- IsCurrentValue